AN EULER SOLVER ACCELERATOR IN FPGA FOR COMPUTATIONAL FLUID DYNAMICS APPLICATIONS

Diego Sanchez-Roman, Gustavo Sutter, Sergio Lopez-Buedo, Ivan Gonzalez,
Francisco J. Gomez-Arribas and Javier Aracil

Escuela Politecnica Superior
Universidad Autonoma de Madrid
{d.sanchez; gustavo.sutter; sergio.lopez-buedo; ivan.gonzalez; francisco.gomez; javier.aracil}@uam.es

ABSTRACT

This paper addresses the problem of accelerating Computational Fluid Dynamics (CFD) applications, utilized by aeronautical engineers to create more efficient and aerodynamic designs. CFD applications require intensive floating point calculations, so they are usually executed on High-Performance Computing (HPC) systems. Here, we study the HW implementation of a cell-vertex finite volume algorithm to solve Euler equations, using the XtremeData XD2000i in-socket FPGA accelerator. Taking advantage of high-level language synthesis tools together with optimized low level components, a HW-accelerated implementation that achieved speedups up to 13.25x could be created in a short time.

1. INTRODUCTION

Computational Fluid Dynamics (CFDs) is a powerful tool used in the design and optimization process of several industrial applications. Specifically, the aeronautical industry is expected to get high benefits from CFDs by replacing the high costly wind tunnel essays with computer simulations. The problem is that CFD algorithms require many iterations to converge and, in general, the space has to be finely discretized in order to obtain accurate results. Because of this, simulations take too much time and cannot be performed as regularly as it would be desired.

In the last years, great effort is being done to accelerate the execution of these algorithms. Among others, computing parallelization [1], GPU computing [2, 3] and FPGA solutions [4, 5] have been developed. Here, we study the acceleration of the Euler 2D algorithm using the XtremeData XD2000i FPGA in-socket accelerator [6].

On the other hand, interest in high level to HDL languages is gaining momentum due to the promise of lower development times and a performance similar to that of hand-coded HDL solutions [7, 8]. In our study, we used the Impulse C software from Impulse Accelerated Technologies [9] which generates VHDL or Verilog from standard ANSI C code with very few limitations.

This paper is organized as follows: Section 2 provides a brief description of the algorithm used to solve Euler equations. Execution time is analyzed and the election of routines to be implemented in HW is justified. Section 3 describes the development system, which basically consists of a dual processor computer where one of the processors is replaced by the XD2000i in-socket accelerator. Section 4 details the HW to SW stream interface. Section 5 analyzes the implementation of the routines being accelerated, discussing the optimization techniques used, the problems found and the solution to those difficulties. Results are presented and discussed in section 6. Finally, section 7 summarizes the future lines of work.

2. EULER ALGORITHM

The target algorithm implements a cell-vertex finite volume method to solve Euler equations, which govern inviscid flows. The space is discretized in an unstructured mesh where the control volumes can be triangles or rectangles in 2D and hexahedrons, pyramids, tetrahedrons or wedges in 3D. The main loop of the program is represented in Fig. 1 and consists of four steps: (1) time-step calculation (2) space-integration (3) time integration and (4) inviscid forces calculation.

Time profiling showed that about 70% of the execution time was spent in steps (2) and (3). In addition to that, the new values for the conservative variables -density, momentum vector and energy- at each node are also calculated in these two steps. So we initially chose routines (2) and (3) to be implemented in hardware. However, once we solved the area restrictions, we also implemented step (1), as it allowed us to perform multiple time iterations without exchanging data with the CPU. In the space-integration routine (2), gradients and residuals are computed. In time integration (3), the new conservative values are calculated using these residuals and the time step. The data dependency for a node, in order to get its new conservative values, consists of the conservative

978-1-4244-8848-3/11/$26.00 ©2011 IEEE

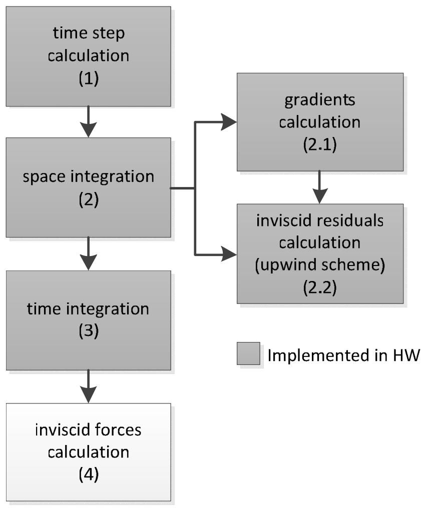
time step calculation (1)
space integration (2)
time integration (3)
inviscid forces calculation (4)
gradients calculation (2.1)
inviscid residuals calculation (upwind scheme) (2.2)
Implemented in HW

Fig. 1. Algorithm Flow

values of its neighbors and neighbors of its neighbors. In addition, geometrical information is also needed, including normal vectors and the length of edges. Lastly, the inviscid forces routine (4) takes the conservative variables computed previously to derive the stress over the boundaries, but it does not modify the node values. In fact, it can be moved out the main loop if there is no need of stress history among time iterations.

It is also relevant to note that the whole main loop is geometrically agnostic. Data connectivity and geometrical computations are computed in a first preprocessing step so it does not matter to the main loop routines how the space domain was discretized. The algorithm extensively uses floating point arithmetic, with more than 250 operations, mostly additions/subtractions and multiplications.

3. DEVELOPMENT SYSTEM

The development system is a computer with a dual quad-core Xeon motherboard populated with one Intel Xeon processor and one XD2000i FPGA Coprocessor Module. The XD2000i, installed in the second Xeon processor socket, uses the motherboard's existing CPU infrastructure to create a full-featured environment for FPGA coprocessor functions. The high-bandwidth, low-latency Front-Side Bus (FSB) link between the XD2000i and the Xeon enables tightly-coupled FPGA acceleration of x86 applications, previously impossible with legacy PCI-based solutions [6].

3.1. Host architecture

The development system contains one Intel Xeon L5408 (a 45 nm quad core processor running at 2.13 GHz with 12 MB of L2 cache). Also, the system features 32 GB of RAM and a FSB running at 1066 MHz.

3.2. Accelerator

The HW module is the XtremeData XD2000i in-socket accelerator [6] which contains three Altera Stratix III FPGAs. This module plugs directly into the FSB socket of any dual processor Xeon system. Hardware integration is as simple as pulling one of the Xeon chips and inserting the pin-compatible XD2000i.

Of the three FPGAs within the XD2000i in-socket accelerator, one serves as bridge to the FSB whereas the other two are available to implement the user logic. These two application FPGAs are connected through two 64-bit dedicated buses. In addition, the XD2000i module includes two QDRII+ SRAM banks, one for each user FPGA. Each of these banks has a size of 8 MB -2MB x 32 bits wide- and has two independent read/write ports with a bandwidth of 2.8 GB/s for each one. The read latency is 2.5 cycles and 4-word burst reads are allowed [6].

3.3. Software configuration

The system runs a 64-bit CentOS Linux distribution. XtremeData provides an open source Linux device driver which allows user space applications to directly access memory mapped FPGA resources, service FPGA interrupt requests, lock physical memory, and perform FPGA-controlled DMAs.

4. HW/SW INTERFACE

Fig. 2 represents our computational unit, the mesh fragment. Nodes are connected through edges, forming the control volumes where physical laws are imposed. Since each node requires two neighborhood layers to compute its new conservative variables, we distinguish between two types of nodes, target and non-targets. Target nodes are those surrounded by two vicinity layers, for which new values of conservative variables are obtained. It is also interesting to identify edges that connect to a target node, first edges, because residuals calculation only requires visiting those edges.

All this information is packed in a stream. There is a first header block that contains the number of nodes, edges, boundary conditions, etc. The distinction between target and non-target nodes is based on the position, target nodes are placed first in the stream. The same approach is utilized to distinguish first edges.

As the interface provided by XtremeData and Impulse C to the application FPGAs consists of two -input and output- 256-bit links at 100 MHz, each element is packed in a 256-bit block. Node information is composed of its conservative values and the associated control volume -5 floating point numbers-. Edges contain the indexes of the connected nodes -16 bit wide each-, the edge vector, and the normal vector. There is a little misuse in the bandwidth

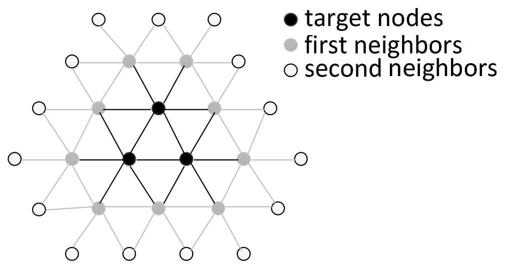

Fig. 2. Mesh Diagram

which is justified because of an easier implementation and an insignificant performance difference. Finally, the output stream contains exclusively the new conservative values of target nodes.

5. IMPLEMENTATION

Writing a VHDL implementation of the selected routines by hand is a huge task and requires very skilled engineers. In the last years, HW synthesis from high level languages has improved enough to allow standard software engineers to produce HW easily [7, 8, 10-12].

In our study, we chose Impulse C to generate VHDL from C code. The changes required from the original code are straightforward and are related to the stream/process paradigm of Impulse C. So, time required to create the HW design is dramatically reduced when compared to a HDL solution from scratch.

However, optimization techniques must be applied in order to achieve good results. These techniques are basically loop unrolling and pipelining. Impulse C provides a smart way of doing these optimizations through the use of compiler directives (*pragmas*), without extra changes in the code.

In the following subsections implementation details are described. Subsection 5.1 is related to trivial changes in the C code in order to save resources. Saving resources is critical in order to have enough silicon to develop performance optimizations. Subsection 5.2 describes the pipeline latency problem due to the high-latency single precision floating point operators provided by Impulse C and Altera, and the solution implemented with our own IEEE-754 low-latency single precision floating point library. Subsection 5.3 describes techniques used to improve pipeline throughput. Finally, Subsection 5.4 includes the HW architecture and the resource utilization of the implementation.

5.1. Trivial C code modification for pipeline latency and resource usage reduction

The main problem with pipelining and loop unrolling is the area increase. Loop unrolling replicates HW to achieve parallelism while pipelining requires registers to propagate data through the pipeline stages.

Because of this extra area requirements, massive use of these optimizations resulted in a design which did not fit in the target FPGA. Thus, we had to restrict these optimizations to the most costly routine: the residuals computation. However, even pipelining just this portion of the algorithm did not fit in the device. Some trivial changes in the C code are helpful to reduce arithmetic operators and pipeline latency. In fact, many of these optimizations would be done automatically by a general SW compiler. However, HW compilers are yet in an early stage of development and we had to look carefully into the code to make the following changes by hand.

1. Reuse of operators: when the same arithmetic operation is performed between two variables with the same values several times in the code, it is convenient to compute this operation to an auxiliary variable and substitute all the operations by it. For example, $a = c + x * y$; $b = c * x * y$ can be replaced by $aux = x * y$; $a = c + aux$; $b = c * aux$. Doing so, one multiplier is saved.
2. Extract common factor: this also saves the number of arithmetic operators needed. For example, in order to compute $a * b + a * c - a * d$, it is necessary the use of three multipliers, one adder and one subtractor. However, the arithmetic equivalent $a * (b + c - d)$ only uses one multiplier.
3. Avoid division when possible: dividers are one of the most expensive HW components. In general they require much more area and have more latency than multipliers. So, if a division between a variable is performed more than once, it can be convenient to compute its inverse and multiply instead.
4. Reuse of auxiliary variables: using the same variable to store auxiliary values can lead to latency penalizations. Impulse C schedules operations in a best-first policy allowing code A wrote after code B to be executed first when there is no data dependence between both codes. In this case, if we use the same auxiliary variable in code A and B, data dependency is generated and code A must be scheduled after code B, increasing pipeline latency unnecessary and increasing the number of registers needed in order to propagate data through the pipeline.

5.2. Reducing pipeline latency with low-latency floating point operators.

Although changes in the code explained in Section 5.1 significantly reduced pipeline latency and operators, area usage was still huge. The design barely fitted and no more optimizations could be done. Residuals calculation pipeline had yet many stages (406) due to the high latency of the floating point operators provided by Impulse C and its intensive use (85 floating point adders/substractors, 95 floating point multipliers, 16 floating point dividers and 7

Table 1. Floating point operations latencies and area.

	Default			Megafunction			Own		
	Lat	ALUT	Reg	Lat	ALUT	Reg	Lat	ALUT	Reg
Add/sub	11	554	632	7	548	308	3	435	139
Mult	11	148	283	5	55	136	2	107	72
Div	15	2127	710	6	1482	758	6	1320	392
Div (DSP)	15	244	491	-	-	-	-	-	-
Sqrt	NA	NA	NA	16	473	521	5	401	198

floating point square roots). Since our design is restricted to 100 MHz, there is no need of such high latency operators and we coded our own low-latency floating point library. Table 1 shows the latencies and area provided by default operators in Impulse C vs. Altera Megafunctions and our own library. In addition to the standard operators, we also coded multiplication and division by two because of its simplicity and the area and latency savings. The integration of our custom VHDL operators was easy thanks to flexibility of Impulse C, and no changes in the C code were needed. In this way, we lowered the latency to 98 stages, while using Megafunction cores would only reduce the latency to 226 stages. This allowed us to unroll and pipeline other loops of the algorithm and perform the optimizations explained in the next section.

5.3. Pipeline throughput optimization

As a result of coding our own floating point library and using the second user FPGA, we get enough space to unroll and pipeline other parts of the algorithm. In a pipeline design, the goal must be to achieve one result per clock cycle. However, we found two problems:

Throughput limited due to memory accesses: Impulse C maps each C array to memory blocks, which can perform up to two parallel reads/writes (dual port memory). In the original C code, there were multidimensional arrays. For example, gradients were stored in a 3D dimensional array (gradients of conservative values for each dimension and for the node). On the other side, nested loops inside a pipeline must be unrolled in Impulse C. In general, this increases the instruction level parallelism. However, this was not possible by using 3D arrays, since a single memory block could not provide the required data parallelism. Therefore, memory had to be accessed multiple times, blocking the next loop iteration. To solve this, we flattened the arrays for each conservative variable and dimension so that multiple memory blocks were inferred, thus providing the desired data parallelism. This required changes in the C code which were minimized by defining macros. However, code legibility was affected.

Throughput limited due to address conflicts in accumulation: edge traveling is performed in order to compute gradients/residuals. For each edge, the computed gradient/residual value must be added or subtracted to the partial gradient/residual of each node. In the edge traveling loop, this creates dependences between iterations. The problem is that the next iteration must wait to the current one to write the partial gradient/residual, in order to avoid the read after write (RAW) hazard. To workaround this, we developed an out-of-order accumulator.

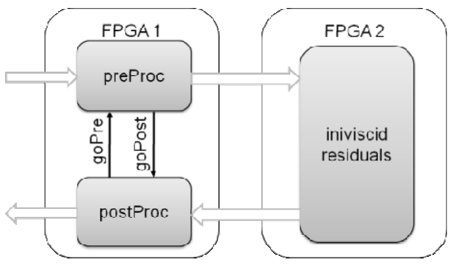

Fig. 3. HW processes distribution

5.4. HW architecture

Routines (1), (2) and (3) are implemented in three processes distributed between both application FPGAs, as shown in Fig. 3.

The preProc process reads and stores the mesh in the FPGA1 block-RAMs. It also performs gradient calculation (2.1). FPGA2 only contains the inviscid residuals calculation process (2.2), which receives its input data from process preProc in FPGA 1. In parallel with gradient computation, postProc process in FPGA 1 executes the local time-step routine (1). Finally, postProc reads residuals and performs the time integration step (3). Also, there are two signals between preProc and postProc for synchronization purposes.

As we noted above, the mesh streamed to the XD2000i in-socket accelerator is stored in the internal memory blocks of the first FPGA. In the current implementation, a maximum of 8192 target and 12288 total nodes can be stored. The maximum number of edges is of 24576. This leads to an 81% of memory usage. Detailed information about resource utilization is shown in Table 2. The first row indicates the available resources for each user FPGA. The following rows indicate the resource utilization of each FPGA.

6. TIME RESULTS AND DISCUSSION

Two test cases were used to compare execution times and validate results. The first one is a coarse discretization of NACA 0012 airfoil while the second is a finer discretization of the same wing section. Table 3 summarizes the meshes characteristics. These meshes translated to the stream format described in Section 4 have a size of 614 KB and 989 KB respectively.

As a first step we analyzed the bandwidth achieved by a simple Impulse C design which only receives data and writes it back. The results obtained are shown in Fig. 4.

Table 2. Resource utilization.

	Comb ALUT	Mem ALUT	Logic Reg	DSP Block	Mem (Mbit)
Total	203520	101760	203520	768	15.040
FPGA 1	86916	7696	97664	288	12.135
FPGA 2	67564	144	124240	628	1.388

Table 3. Meshes used for testing

Mesh	Nodes	Edges	Triangles
NACA_2D_coarse	5233	15449	10216
NACA_2D_fine	7908	23296	15388

Then, the time needed to send and receive the test meshes would be 1 ms and 1.5 ms approximately. It must be pointed out that XtremeData is developing a new version of the HW which communicates with the FSB as a master, so great improvements in bandwidth are expected.

Execution time is summarized in Table 4. The SW-based solution is run in the development system described in Section 3 and it is compiled with gcc 4.1.2 and maximum compilation effort, -O3. CPU values consist of the execution time of routines (1), (2) and (3), for one iteration. The XD2000i results include transmission of the meshes, computation and reception of the new conservative variables through the FSB. In parentheses, the execution time is separated in transfer and computation time. The transfer time is derived from the discussion below. Since the two test cases fit in the FPGA block-RAMs, the results from one iteration can be used to compute another one without exchanging data with the host. Consequently, greater speedups are achieved. Table 5 shows the execution time per iteration when 100 iterations are performed in the HW, neglecting data transfers. Also, with these results we can infer that the transfer times for the grids are of 0.8 ms and 1.1 ms. The difference in the transfer times analyzed previously is explained because the data returned to the host is much smaller as it only consists of the conservative values of the target nodes. In a more realistic problem, the whole mesh may not fit in the FPGA memories and partitioning has to be performed. In this case, it is necessary to transfer the results to the host in each iteration so that the values of the two vicinity layers -which are computed in other fragments- are updated. At this point, the results displayed in Fig. 5 come into play. As it turns out, it can be of interest to send two or four extra vicinity layers with each fragment so that the HW can iterate two or three times without transferring data to the host.

6.1. Result comparisons

It is hard to compare with previous approaches, since there is no standard benchmark for aeronautical CFD applications. In order to put our work in context, we will mention some of the most relevant previous works. [13] Solves the 1D problem using the FPGA to calculate Euler

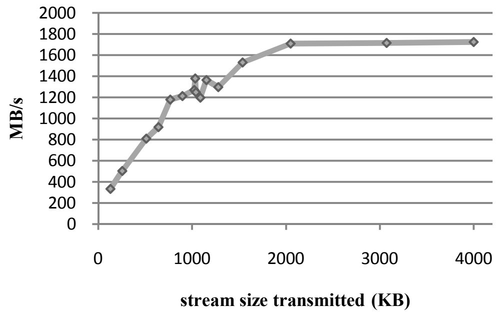

Fig. 4. Bandwidth to XD2000i in-socket accelerator

and Roe's flux, they report a speed up of 1.31 in Virtex-4 compared to a software solution. They speculate that using a 32 Virtex-4 board and integrating 20 cores in each FPGA they could accelerate two orders of magnitude. [4] Proposes a systolic architecture to solve 2D problems. Using Altera Stratix 2 and for small square cavity of 25 by 25 nodes they report a speedup factor of 7.14 respect to a Pentium 4 at 3.2 GHz. [14] accelerates the CFD method of the Japan aerospace agency using several Virtex-4 devices. The authors claim that the parts of the algorithm accelerated in hardware are 170 times and 46.8 times faster, but disclaims that this only represent 23.9% of the total computation time. In contrast, our solution covers the full algorithm with one order of magnitude performance speedup.

6.2. Future work discussion

We have started analyzing the computation over bigger meshes. Partition techniques have to be used since these more realistic problems do not fit in the internal FPGA memory. Due to the minimum two-layer vicinity required for each fragment, there will be redundant data between the fragments. In order to reduce this overhead, graph partition techniques must be explored to achieve connected fragments of similar sizes. Related to the previous point, we will explore the use of the QDRII memory banks since they will allow processing bigger fragments. Also, we will deal with the SW-HW integration, i.e., how to assemble the returned values from the FPGA in the original SW structures and how to update the streams of each fragment in order to compute subsequent time iterations. Our goal will be to overlap the computation and the update of the fragments to avoid performance penalization.

Also, we will finish the implementation of the Euler 3D version. In fact, we currently have a non-optimized version running. The differences between the 2D and 3D versions reduce basically to the addition of a new conservative variable, the z-momentum, but the whole algorithm flow is exactly the same.

Another line of work will be the implementation of a 2D Navier-Stokes version of the current algorithm. An initial analysis showed it is more computationally

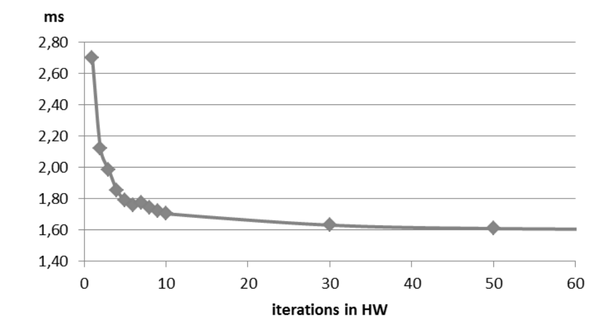

Fig. 5. Mean time per iteration against number of iterations in HW

Table 4. Time execution for one iteration

Mesh	CPU (ms)	XD2000i (ms)	Speed up
NACA_2D_coarse	10.1	1.8 (0.8+1.0)	5.61
NACA_2D_fine	21.2	2.7 (1.1+1.6)	7.85

Table 5. Time execution per iteration, 100 iterations

Mesh	CPU (ms)	XD2000i (ms)	Speed up
NACA_2D_coarse	10.1	1.0	10.10
NACA_2D_fine	21.2	1.6	13.25

expensive. However, it seems that inviscid and viscous residuals computation can be parallelized so greater accelerations can be achieved. However, area challenges will arise.

Finally, we will investigate the analysis of the power efficiency of our design, since FPGAs are well known to be much more less power hungry than CPUs and GPUs.

7. CONCLUSIONS

A cell-vertex finite volume method to solve the 2D Euler equations has been implemented in the XD2000i in-socket FPGA accelerator. HW has been generated from C code, using Impulse C tools, obtaining one order of magnitude speedups. Development time is reduced thanks to the use of high level language synthesis tools. However, a full understanding of the low level HW was required in order to tune up the implementation. Low-latency floating point operators were developed in order to save resources and an out-of-order module was designed to improve performance. A greater speedup is expected with the implementation of the Navier-Stokes algorithm.

Acknowledgment: This project was granted by DOVRES project (FGUAM ref 059901), part of FuSim-E initiative.

8. REFERENCES

[1] L. Xiao, X. Zhang, Z. Kuang, B. Feng, and J. Kang, "Auto-CFD: efficiently parallelizing CFD applications on clusters," in *Proc. IEEE Int Cluster Computing Conf*, 2003, pp. 46–53.

[2] T. Brandvik and G. Pullan, "Acceleration of a 3D Euler solver using commodity graphics hardware," in *46th AIAA Aerospace Sciences Meeting and Exhibit*, 2008.

[3] I. Kampolis, X. Trompoukis, V. Asouti, and K. Giannakoglou, "CFD-based analysis and two-level aerodynamic optimization on graphics processing units," *Computer Methods in Applied Mechanics and Engineering*, 2009.

[4] K. Sano, T. Iizuka, and S. Yamamoto, "Systolic Architecture for Computational Fluid Dynamics on FPGAs," in *Proc. 15th Annual IEEE Symp. Field-Programmable Custom Computing Machines FCCM 2007*, 2007, pp. 107–116.

[5] W. Smith and A. Schnore, "Towards an RCC-based accelerator for computational fluid dynamics applications," *The journal of Supercomputing*, vol. 30, no. 3, pp. 239–261, 2004.

[6] XtremeData XD2000i in-socket accelerator, http://www.xtremedata.com/products/accelerators/in-socket-accelerator/xd2000i.

[7] J. Curreri, S. Koehler, B. Holland, and A. D. George, "Performance Analysis with High-Level Languages for High-Performance Reconfigurable Computing," in *Proc. 16th Int. Symp. Field-Programmable Custom Computing Machines FCCM '08*, 2008, pp. 23–30.

[8] T. Dedek, T. Martinek, and T. Marek, "High Level Abstraction Language as an Alternative to Embedded Processors for Internet Packet Processing in FPGA," in *Proc. Int. Conf. Field Programmable Logic and Applications FPL 2007*, 2007, pp. 648–651.

[9] Impulse C, http://www.impulseaccelerated.com.

[10] J. J. Koo, D. Fernandez, A. Haddad, and W. J. Gross, "Evaluation of a High-Level-Language Methodology for High-Performance Reconfigurable Computers," in *Proc. ASAP Application -specific Systems, Architectures and Processors IEEE Int. Conf*, 2007, pp. 30–35.

[11] J. J. Lopes, J. S. Luiz, E. Marques, and J. M. P. Cardoso, "A Benchmark Approach for Compilers in Reconfigurable Hardware," in *Proc. 6th Int System-on-Chip for Real-Time Applications Workshop*, 2006, pp. 120–124.

[12] K. Torkelsson and J. Ditmar, "Header compression in Handel-C-an Internet application and a new design language," in *Proc. Euromicro Symp. Digital Systems, Design*, 2001, pp. 2–7.

[13] E. Andres, M. Molina, G. Botella, A. del Barrio, and J. Mendias, "Aerodynamics analysis acceleration through reconfigurable hardware," in *Proc. 4th Southern Conf. Programmable Logic*, 2008, pp. 105–110.

[14] H. Morishita, Y. Osana, N. Fujita, and H. Amano, "Exploiting memory hierarchy for a computational fluid dynamics accelerator on fpgas," in *Proc. Int. Conf. ICECE Technology FPT 2008*, 2008, pp. 193–200.